A Gentle Adventure Mechanising Message Passing Concurrency Systems

An Experience/Walkthrough Report

David Castro-Perez, Lorenzo Gheri, Francisco Ferreira, Martin Vassor, and Nobuko Yoshida

DisCoTec22

Roadmap

1. Binders and Linearity

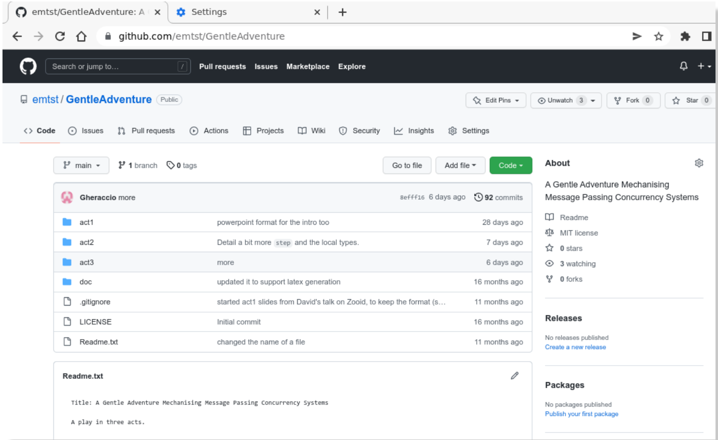
Act I: Binary Session Types

2. Multiparty Processes and Coinduction

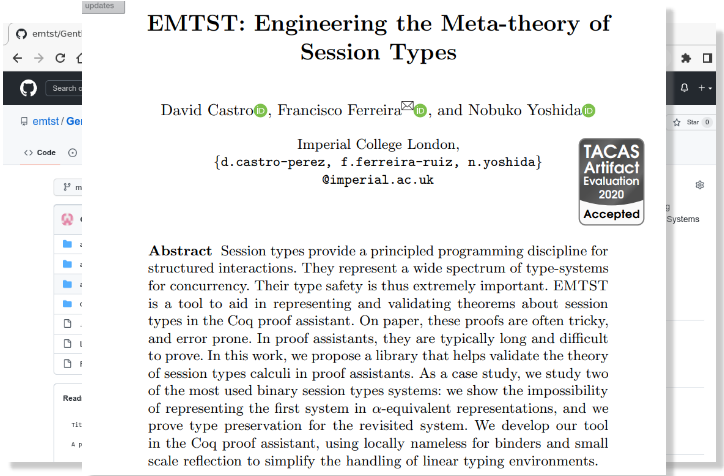
Act II: Mechanising Multiparty Processes

Act III: Mechanising Multiparty Session Types

SmolEMTST: Tutorial Repository



SmolEMTST: Tutorial Repository



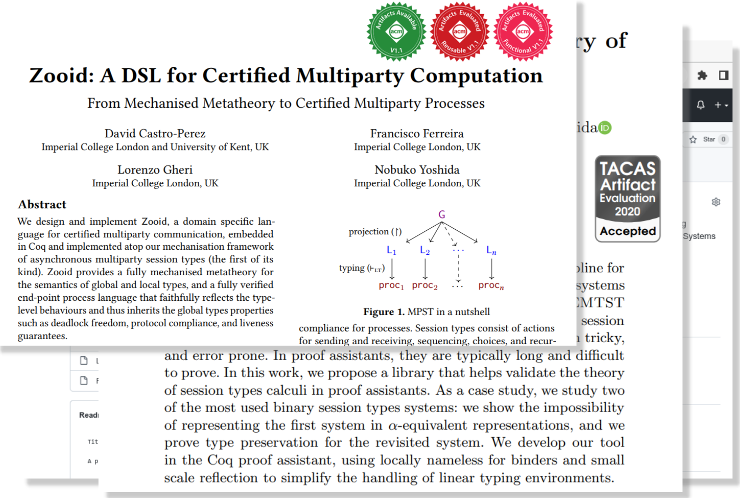
EMTST: Engineering the Meta-theory of Session Types

David Castro, Francisco Ferreira, and Nobuko Yoshida

Imperial College London,
{d.castro-perez, f.ferreira-ruiz, n.yoshida}
@imperial.ac.uk

Abstract Session types provide a principled programming discipline for structured interactions. They represent a wide spectrum of type-systems for concurrency. Their type safety is thus extremely important. EMTST is a tool to aid in representing and validating theorems about session types in the Coq proof assistant. On paper, these proofs are often tricky, and error prone. In proof assistants, they are typically long and difficult to prove. In this work, we propose a library that helps validate the theory of session types calculi in proof assistants. As a case study, we study two of the most used binary session types systems: we show the impossibility of representing the first system in α-equivalent representations, and we prove type preservation for the revisited system. We develop our tool in the Coq proof assistant, using locally nameless for binders and small scale reflection to simplify the handling of linear typing environments.

SmolEMTST: Tutorial Repository



Mechanising the **Honda, Vasconcelos and Kubo's binary session type system**

Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In Hankin, C., editor, Programming Languages and Systems, pages 122–138, Berlin, Heidelberg. Springer Berlin Heidelberg

Processes: Key Features

$$
\begin{array}{rcll}
P, Q, R & ::= & & \\
& \mid & k\,![e];\; P & \text{data sending} \\
& \mid & k\,?(x).P & \text{data receiving} \\
& \mid & \mathtt{throw}\; k\,[k'];\, P & \text{channel sending} \\
& \mid & \mathtt{catch}\; k\,(k').P & \text{channel receive} \\
& \mid & P \mid Q & \text{parallel composition} \\
& \mid & \nu_c\,(k).P & \text{channel hiding} \\
& \mid & \ldots &
\end{array}
$$

Processes: Key Features

$$
\begin{array}{rcll}
P, Q, R & ::= & & \\
& \mid & k\,![e];\ P & \text{data sending} \\
& \mid & k\,?(x).P & \text{data receiving} \\
& \mid & \texttt{throw}\ k\,[k'];P & \text{channel sending} \\
& \mid & \texttt{catch}\ k\,(k').P & \text{channel receive} \\
& \mid & P \mid Q & \text{parallel composition} \\
& \mid & \nu_c\,(k).P & \text{channel hiding} \\
& \mid & \ldots &
\end{array}
$$

Binder Mechanisation: DeBruijn Indices

Terms $\quad M, N ::= n \mid M\ N \mid \lambda.\ M$

Binder Mechanisation: DeBruijn Indices

Natural number

Terms $\quad M, N ::= n \mid M\, N \mid \lambda.\, M$

Binder Mechanisation: DeBruijn Indices

Terms $\quad M, N ::= n \mid M\,N \mid \lambda.\,M$

$$\lambda.\,(\lambda.\,0)\,(\lambda.\,0\;1)$$

Binder Mechanisation: Locally Nameless

Natural number

Name that represents a free variable

Terms: $M, N ::= n \mid x \mid M\,N \mid \lambda.\,M$

Binder Mechanisation: Locally Nameless

Terms: $M, N ::= n \mid x \mid M\ N \mid \lambda.\ M$

$M^x \equiv \{0 \to x\}M$ Open a term

$\backslash^x M \equiv \{0 \leftarrow x\}M$ Close a term

$\mathsf{lc}(M)$ Locally closed term

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\forall x \notin L \qquad \Gamma, x : S \vdash M^x : T}{\Gamma \vdash \lambda.\ M : S \to T}$$

$$\frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash M\ N : T}$$

Process Mechanisation

```
Inductive name : Set :=
  | fnm : atom → name
  | bnm : nat → name
.

Definition channel := name.

Inductive proc : Set :=
| send : channel → exp → proc → proc
| receive : channel → proc → proc

| throw : channel → channel → proc → proc
| catch : channel → proc → proc

| nu_ch : proc → proc (* hides a channel *)
...
```

```
Inductive lc : proc → Prop :=
| lc_send : forall k e P,
    lc_nm k →
    lc_exp e →
    lc P →
    lc (send k e P)

| lc_receive : forall (L : seq atom) k P,
    lc_nm k →
    (forall x, x \notin L → lc (open P x)) →
    lc (receive k P)


| lc_throw : forall k k' P,
    lc_nm k → lc_nm k' →
    lc P →
    lc (throw k k' P)

| lc_catch : forall (L : seq atom) k P,
    lc_nm k →
    (forall x, x \notin L → lc (open P x)) →
    lc (catch k P)
...
```

Semantics (Excerpt as in Paper)

$$
\begin{array}{rl}
\textsc{R-Pass} & \texttt{throw}\ k\,[k'];P \mid \texttt{catch}\ k\,(k').Q \rightarrow P \mid Q \\
\textsc{R-Com} & k\,![e];\ P \mid k\,?(x).Q \rightarrow P \mid \{x \rightarrow e\}Q \\
\textsc{R-Cong} & P \equiv P' \text{and } P' \rightarrow Q' \text{and } Q' \equiv Q \Rightarrow P \rightarrow Q \\
\textsc{R-Scop} & P \rightarrow Q \Rightarrow \nu_c\,(k).P \rightarrow \nu_c\,(k).Q \\
\textsc{R-Par} & P \rightarrow P' \Rightarrow P \mid Q \rightarrow P' \mid Q
\end{array}
$$

Mechanising the Semantics

Using Locally Nameless, α-equivalent terms are **syntactically equal**

How do we mechanise R-PASS?

$$\texttt{throw}\ k\,[k'];P \mid \texttt{catch}\ k\,(k').Q \rightarrow P \mid Q$$

Mechanising the Semantics

Using Locally Nameless, α-equivalent terms are **syntactically equal**

How do we mechanise R-PASS?

$$\texttt{throw}\ k\,[k'];P \mid \texttt{catch}\ k\,(k').Q \to P \mid Q$$

A naive attempt:

$$\texttt{throw}\ k\,[k'];P \mid \texttt{catch}\ k\,().Q \to P \mid Q^{k'}$$

Mechanising the Semantics

Using Locally Nameless, α-equivalent terms are **syntactically equal**

How do we mechanise R-PASS?

$$\texttt{throw}\ k\,[k'];P \mid \texttt{catch}\ k\,(k').Q \to P \mid Q$$

A **WRONG** attempt:

$$\texttt{throw}\ k\,[k'];P \mid \texttt{catch}\ k\,().Q \to P \mid Q^{k'}$$

Why is our mechanisation of r-pass wrong?

$$\begin{array}{lrcl} \text{Type} & \alpha, \beta & ::= & ![S];\alpha \mid ?[S];\alpha \mid \mathbf{end} \mid \bot \\ \text{Typing} & \Delta & ::= & \cdot \mid \Delta, k : \alpha \end{array}$$

Subject Reduction: if $\Gamma \vdash P \triangleright \Delta$ with Δ balanced, and $P \to Q$, then there exists Δ' s.t. $\Gamma \vdash Q \triangleright \Delta'$, with Δ' balanced.

Why is our mechanisation of r-pass wrong?

$$\begin{array}{lrcl} \text{Type} & \alpha, \beta & ::= & ![S];\alpha \mid ?[S];\alpha \mid \mathbf{end} \mid \bot \\ \text{Typing} & \Delta & ::= & \cdot \mid \Delta, k : \alpha \end{array}$$

Subject Reduction: if $\Gamma \vdash P \triangleright \Delta$ with Δ balanced, and $P \to Q$, then there exists Δ' s.t. $\Gamma \vdash Q \triangleright \Delta'$, with Δ' balanced.

Why is our mechanisation of r-pass wrong?

$$\begin{array}{lrcl} \text{Type} & \alpha, \beta & ::= & ![S]; \alpha \mid ?[S]; \alpha \mid \mathbf{end} \mid \bot \\ \text{Typing} & \Delta & ::= & \cdot \mid \Delta, k : \alpha \end{array}$$

Subject Reduction: if $\Gamma \vdash P \triangleright \Delta$ with Δ balanced, and $P \rightarrow Q$, then there exists Δ' s.t. $\Gamma \vdash Q \triangleright \Delta'$, with Δ' balanced.

Rule $\texttt{throw}\ k\,[k'];P \mid \texttt{catch}\ k\,().Q \rightarrow P \mid Q^{k'}$
breaks subject reduction.

The Problem with Equating α-equivalent Terms

The idea behind R-PASS:

$$\texttt{throw}\ k\,[k_0];P \mid \texttt{catch}\ k\,(k_1).Q \to P \mid Q'$$
$$\text{if} \quad \texttt{catch}\ k\,(k_1).Q \equiv_\alpha \texttt{catch}\ k\,(k_0).Q'$$

The Problem with Equating α-equivalent Terms

The idea behind R-PASS:

$$\begin{array}{l} \texttt{throw}\ k\,[k_0]; P \mid \texttt{catch}\ k\,(k_1).Q \to P \mid Q' \\ \text{if} \quad \texttt{catch}\ k\,(k_1).Q \equiv_\alpha \texttt{catch}\ k\,(k_0).Q' \end{array}$$

k_0 cannot be free in Q

The Problem with Equating α-equivalent Terms

The idea behind R-PASS:

$$\begin{array}{l}\texttt{throw}\ k\,[k_0];P \mid \texttt{catch}\ k\,(k_1).Q \rightarrow P \mid Q' \\ \text{if}\quad \texttt{catch}\ k\,(k_1).Q \equiv_\alpha \texttt{catch}\ k\,(k_0).Q'\end{array}$$

k_0 cannot be free in Q

We used a **standard** representation of binders, to mechanise a **non-standard, but correct** use of binders.

Mechanising the Revised System

Yoshida, N. and Vasconcelos, V. T. (2007). Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. Electronic Notes in Theoretical Computer Science, 171(4):73 – 93. Proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006)

Processes, Channels, and Polarities

A solution to the naive R-PASS is to distinguish **channel polarities** ([Gay and Hole, 2005])

$$p, q ::= + \mid -$$
$$\texttt{throw}\ k^{p}\,[k'^{q}]; P \mid \texttt{catch}\ k^{\overline{p}}\,().Q \to P \mid Q^{k'^{q}}$$

```
Inductive channel :=
| Ch of (kvar * polarity)
| Var of cvar.

Inductive proc : Set :=
| send : channel → exp → proc → proc
| receive : channel → proc → proc

| throw : channel → channel → proc → proc
| catch : channel → proc → proc

| nu_ch : proc → proc
...
```

Atoms: Separating Namespaces

```
Module CA := AtomScope Atom.Atom. (* Module of the atoms for channels *)
Module KA := AtomScope Atom.Atom. (* Module of the atoms for channel name *)
Module EA := AtomScope Atom.Atom. (* Module of the atoms for expressions *)

Notation cvar := (CA.var).
Notation kvar := (KA.var).
Notation evar := (EA.var).
```

Typing

Type $\alpha, \beta \quad ::= \quad ![S]; \alpha \mid ?[S]; \alpha \mid \mathbf{end} \mid \ldots$

Typing $\Delta \quad ::= \quad \cdot \mid \Delta, k : \alpha$

Judgement $\Gamma \vdash P : \Delta$

```
Inductive tp : Set :=
  | input : sort → tp → tp
  | output : sort → tp → tp
  | ended : tp
  | ...
```

```
Inductive oft : sort_env → proc → tp_env → Prop :=
| t_send : forall G kt e P D S T,
    oft_exp G e S →
    oft G P (add kt T D) →
    oft G (send (chan_of_entry kt) e P) (add kt (output S T) D)
| ...
```

Linear Environments

$$\Delta ::= \cdot \mid \Delta, k : \alpha$$

```
Inductive env := Undef | Def of {finMap K → V}.

Definition add x t E :=
  if x \in dom E then Undef else upd x t E.
```

We defined operations on environments that contain **linear** channels.

Adding a channel that is already in the environment results in an **undefined** environment.

We use this fact pervasively in our mechanised proofs.

Subject Reduction

```
Theorem SubjectReductionStep G P Q D:
  oft G P D → P ⟶ Q → exists D', D ⇝ D' ∧ oft G Q D'.
```

```
Theorem SubjectReduction G P Q D:
  oft G P D → P ⟶* Q → exists D', oft G Q D'.
```

Mechanising a Proof of Subject Reduction

Using separate namespaces requires us to prove distinct **substitution lemmas** for every different kind of binder (expression, channel, shared channel).

Separate namespaces helps us avoid errors (e.g. using a channel instead of an expression), and simplifies proofs.

Linear environments allow us to make simplifying assumptions about defined environments.

Summary of Act I

- Deep Embedding binders allows us to fully control the calculus.
- LN requires a number of theorems and lemmas to prove our basic safety properties.
- EMTST (our tool) helps with nominal sets and environments.
- Next, we will explore what do we gain if we give up control (using shallow embeddings).

- Gay, S. and Hole, M. (2005). Subtyping for Session Types in the Pi Calculus. Acta Informatica, 42(2):191–225.
- Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In Hankin, C., editor, Programming Languages and Systems, pages 122–138, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Yoshida, N. and Vasconcelos, V. T. (2007). Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. Electronic Notes in Theoretical Computer Science, 171(4):73 – 93. Proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006).

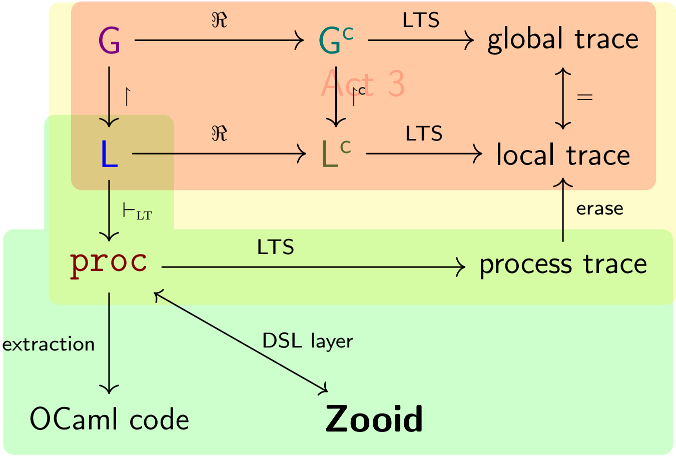
Act II

Smol-Zooid: multiparty with shallower embedding

Goals

1. Certifying **individual** processes of a **distributed system**
2. Extracting runnable code
3. Avoiding complex formalisations of binders, whenever possible

Overview



G
ℜ
G^c
LTS
global trace
↾
Act 3
↾c
=
L
ℜ
L^c
LTS
local trace
⊢LT
erase
proc
LTS
process trace
extraction
DSL layer
OCaml code
Zooid

Smol Zooid

- We combine **shallow/deep embeddings** of binders
  - Processes are defined inductively
  - Values are standard Gallina values
  - We use DeBruijn indices for the deeply embedded binders
- SZooid constructs are **well-typed by construction**

- We leverage **Coq code extraction** mechanism
- For simplicity, SZooid does not cover choices

Core Processes

In: `http://github.com/emtst/gentleAdventure`

```
Inductive proc :=
| Inact | Rec (e : proc) | Jump (X : nat)
| Send (p : participant) {T : type}
     (x : interp_type T) (k : proc)
| Recv (p : participant) {T : type}
    (k : interp_type T -> proc).
```

Payload Types

We need to define a type for payload types:

- We need a decidable equality on payload types
- We need a decidable equality on payload values

```
Inductive type := Nat | Bool | ...
Definition interp_type : type -> Type := ...
```

Semantics: overview

$$P \xrightarrow{E} P'$$

- What is an event?
- How to manage recursion?

Semantics: events

The semantics is an LTS:

- the labels are the **communication events**
- it is parameterised by a **payload interpretation function**
- traces are obtained as the greatest fixpoint of the LTS step

```
Inductive action := a_send | a_recv.
Record event interp_payload :=
  { action_type  : action;
    subj         : participant;
    party        : participant;
    payload_type : type;
    payload      : interp_payload payload_type }.
```

Semantics: Recursion Variables

`p_unroll` exposes the first communication action in a process (unfold recursion):

```
Definition p_unroll e :=
match e with
| Rec e' => p_subst 0 e e'
| e' => e'
end.
```

Semantics: Recursion Variables

```
Fixpoint p_subst d e' e :=
match e with
| Rec e => Rec (p_subst d.+1 e' e)
| Jump X => if X == d then p_shift d 0 e' else e
| ...
end.

Example ex_p_subst:
p_subst 0 (ping_Alice) (Rec (Jump 1)) = Rec ping_Alice.
```

Semantics: recursion unrolling

```
(* unroll uT. Alice!0. T to Alice!0. uT. Alice!0. T *)
Example ex_p_unroll:
p_unroll (Rec (@Send Alice Nat 0 (Jump 0)))
= @Send Alice Nat 0 (Rec (@Send Alice Nat 0 (Jump 0))).
```

Semantics: step

The step of the LTS is defined as a **function**:

```
Definition step' e E :=
  match e with
  | Send p T x k =>
    if (action_type E == a_send) && (party E == p) &&
       (eq_payload (payload E) x)
    then Some k else None
  | Recv p T k => ...  | _ => None
  end.
Definition step e := step' (p_unroll e).
```

Semantics: step

```
Definition event_alice: event interp_type :=
{| action_type := a_send;
   from := Bob;
   to := Alice;
   payload_type := Nat;
   payload := 0 |}.
Example ex_step: step infinite_ping_Alice event_alice
                = Some infinite_ping_Alice.
```

Local Types

- Local types for processes
- Notion of “Being well-typed”
- Simultaneous construction of processes & well-typeness proof

Local Types

```
Inductive lty :=
  | l_end
  | l_jump (X : nat)
  | l_rec (k : lty)
  | l_send (p : participant) (T : type) (l : lty)
  | l_recv (p : participant) (T : type) (l : lty).
```

Type System

```
Inductive of_lty : proc -> lty -> Prop :=
| lt_Send     p T k L x :
    of_lty k L -> of_lty (@Send p T x k) (l_send p T L)
| ...
.

Example ex_of_lty:
of_lty infinite_ping_Alice
(l_rec (l_send Alice Nat (l_jump 0))).
```

Smol Zooid: Smart Constructors

- It would be tedious to type up both a local type and a process
- Users would need to provide a proof that processes are well-typed

We define **SZooid** (Smol Zooid), to write well-typed processes by construction, avoiding repetition.

Smol Zooid: Smart Constructors

```
Definition SZooid L := { p | of_lty p L}.

Definition z_Send  p T x L (k : SZooid L)
  : SZooid (l_send p T L)
  := exist _ _ (lt_Send p x (proj2_sig k)).
...
```

Smol Zooid: Smart Constructors

```
Example Alice_smart_constructor :=
        z_Rec (z_Send Bob Nat 42 (z_Jump 0)).
(*
Alice_smart_constructor:
        SZooid (l_rec (l_send Bob Nat (l_jump 0)))
*)
```

Conclusion

1. Define processes and local types
2. Semantics of processes
3. Automatic construction of local types

- We also have *code extraction*
- and *subject reduction*

Coq-Metatheory for Smol-Zooid

A Gentle Adventure Mechanising Message Passing Concurrency Systems, Act 3

David Castro-Perez, Francisco Ferreira, **Lorenzo Gheri**, Martin Vassor, and Nobuko Yoshida

DisCoTec 2022
Lucca, 13-06-2022

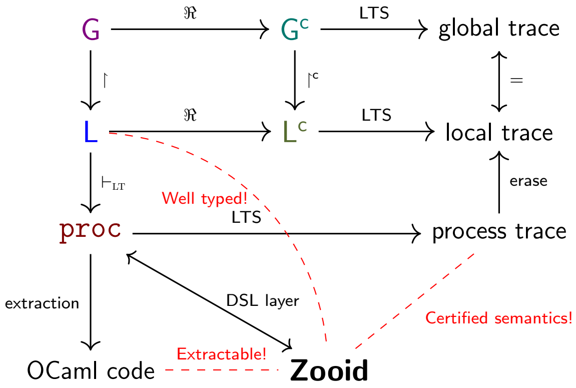
Imperial College London
University of Kent
Royal Holloway University of London

The MPST World, as We Know It

$$
\begin{tikzcd}
 & & \mathsf{G} \arrow[dll, "\text{projection } (\upharpoonright)"'] \arrow[dl] \arrow[d, dashed] \arrow[dr] & \\
\mathsf{L}_1 \arrow[d, "\text{typing } (\vdash_{\mathrm{LT}})"'] & \mathsf{L}_2 \arrow[d] & \cdots \arrow[d, dashed] & \mathsf{L}_n \arrow[d] \\
\mathtt{proc}_1 & \mathtt{proc}_2 & \cdots & \mathtt{proc}_n
\end{tikzcd}
$$

K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. POPL '08

Zoid



D. Castro-Perez, F. Ferreira, L. Gheri, and N. Yoshida. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. PLDI 2021

Introducing the Metatheory of Smol-Zooid Types

$$
\begin{tikzcd}
\mathsf{G} \arrow[r, "\mathfrak{R}"] \arrow[d, "\upharpoonright"] & \mathsf{G}^{\mathsf{c}} \arrow[r, "\mathsf{LTS}"] \arrow[d, "\upharpoonright^{\mathsf{c}}"] & \text{global trace} \arrow[d, leftrightarrow, "="] \\
\mathsf{L} \arrow[r, "\mathfrak{R}"] & \mathsf{L}^{\mathsf{c}} \arrow[r, "\mathsf{LTS}"] & \text{local trace}
\end{tikzcd}
$$

- unravelling preserves projection; focus on coinduction (1^{st} square)
- trace equivalence; focus on soundness (2^{nd} square)

`https://github.com/emtst/GentleAdventure/act3`

Formalisation of Global and Local Types

Inductively Defined Datatypes

$$\begin{aligned} \mathsf{G} \quad ::= &\ \mathtt{end} \\ &\mid X \\ &\mid \mu X.\mathsf{G} \\ &\mid \mathsf{p} \to \mathsf{q} :(\mathbb{S}).\mathsf{G} \end{aligned}$$

$$\begin{aligned} \mathsf{L} \quad := &\ \mathtt{end} \\ &\mid X \\ &\mid \mu X.\mathsf{L} \\ &\mid ![\mathsf{q}];(\mathbb{S}).\mathsf{L} \\ &\mid ?[\mathsf{p}];(\mathbb{S}).\mathsf{L} \end{aligned}$$

Coinductively Defined Datatypes

$$\begin{aligned} \mathsf{G}^{\mathsf{c}} ::= &\quad \mathtt{end}^{\mathsf{c}} \\ &\mid \mathsf{p} \to \mathsf{q} :(\mathbb{S}).\mathsf{G}^{\mathsf{c}} \\ &\mid \mathsf{p} \rightsquigarrow \mathsf{q} :(\mathbb{S}).\mathsf{G}^{\mathsf{c}} \end{aligned}$$

$$\begin{aligned} \mathsf{L}^{\mathsf{c}} ::= &\quad \mathtt{end}^{\mathsf{c}} \\ &\mid !^{\mathsf{c}}[\mathsf{p}];(\mathbb{S}).\mathsf{L}^{\mathsf{c}} \\ &\mid ?^{\mathsf{c}}[\mathsf{q}];(\mathbb{S}).\mathsf{L}^{\mathsf{c}} \end{aligned}$$

Formalisation of Global and Local Types

$$
\begin{tikzcd}
\mathsf{G} = \mu X.\mathsf{p} \to \mathsf{q} :(\mathbb{S}).X \arrow[r, "\Re"] \arrow[d, "\upharpoonright"] & \mathsf{G}^{\mathsf{c}} = \mathsf{p} \to \mathsf{q} :(\mathbb{S}).\mathsf{G}^{\mathsf{c}} \arrow[d, "\upharpoonright^{c}"] \\
\begin{array}{l} \mathsf{G} \upharpoonright_{\mathsf{p}} = \mu X.![\mathsf{q}];(\mathbb{S}).X \\ \mathsf{G} \upharpoonright_{\mathsf{q}} = \mu X.?[\mathsf{p}];(\mathbb{S}).X \end{array} \arrow[r, "\Re"] & \begin{array}{l} \mathsf{L^c}_{\mathsf{p}} = !^{\mathsf{c}}[\mathsf{q}];(\mathbb{S}).\mathsf{L^c}_{\mathsf{p}} \\ \mathsf{L^c}_{\mathsf{q}} = ?^{\mathsf{c}}[\mathsf{p}];(\mathbb{S}).\mathsf{L^c}_{\mathsf{q}} \\ \text{with } \mathsf{G}^{\mathsf{c}} \upharpoonright^{\mathsf{c}}\mathsf{p}\ \mathsf{L^c}_{\mathsf{p}} \\ \text{and } \mathsf{G}^{\mathsf{c}} \upharpoonright^{\mathsf{c}}\mathsf{q}\ \mathsf{L^c}_{\mathsf{q}} \end{array}
\end{tikzcd}
$$

Abandoning Inductive Datatypes

Theorem (Unravelling preserves projections)

Given G, L, G^c *and* L^c, *such that* (a) $G{\upharpoonright}r = L$, (b) $G \mathfrak{R} G^c$, *and* (c) $L \mathfrak{R} L^c$, *then* $G^c \upharpoonright^c r\ L^c$.

$$\begin{tikzcd}
G \arrow[r, "\mathfrak{R}"] \arrow[d, "\upharpoonright"] & G^c \arrow[d, "\upharpoonright^c"] \\
L \arrow[r, "\mathfrak{R}"] & L^c
\end{tikzcd}$$

Proof.

By coinduction. :) □

The Paco Library for Coq: https://plv.mpi-sws.org/paco/

Abandoning Inductive Datatypes

Theorem (Unravelling preserves projections)

Given G, L, G^{c} *and* L^{c}, *such that* (a) $\mathsf{G}\!\upharpoonright\!\mathsf{r} = \mathsf{L}$, (b) $\mathsf{G}\,\Re\,\mathsf{G}^{\mathsf{c}}$, *and* (c) $\mathsf{L}\,\Re\,\mathsf{L}^{\mathsf{c}}$, *then* $\mathsf{G}^{\mathsf{c}} \upharpoonright^{\mathsf{c}}\!\mathsf{r}$ L^{c}.

$$\begin{tikzcd}
\mathsf{G} \arrow[r, "\Re"] \arrow[d, "\upharpoonright"] & \mathsf{G}^{\mathsf{c}} \arrow[d, "\upharpoonright^{\mathsf{c}}"] \\
\mathsf{L} \arrow[r, "\Re"] & \mathsf{L}^{\mathsf{c}}
\end{tikzcd}$$

$\longrightarrow$ Coq!

Proof.

By coinduction. :) $\square$

The Paco Library for Coq: https://plv.mpi-sws.org/paco/

Type Semantics for Zooid

$$\begin{tikzcd} \mathsf{G}^{\mathsf{c}} \arrow[r, "\mathsf{LTS}"] \arrow[d, "\restriction^{\mathsf{c}}"] & \text{global trace} \arrow[d, leftrightarrow, "="] \\ \mathsf{L}^{\mathsf{c}} \arrow[r, "\mathsf{LTS}"] & \text{local trace} \end{tikzcd}$$

With Love, from p to q

p sends:

$$\begin{tikzcd}
\mathsf{p} \to \mathsf{q} : (\mathbb{S}).\mathsf{G}^{\mathsf{c}} \arrow[r, "!\mathsf{pq}\mathbb{S}"] \arrow[d, "\upharpoonright_{\mathsf{p}}"] & \mathsf{p} \rightsquigarrow \mathsf{q} : (\mathbb{S}).\mathsf{G}^{\mathsf{c}} \arrow[r, "?\mathsf{qp}\mathbb{S}"] \arrow[dr, "\upharpoonright_{\mathsf{p}}"] & \mathsf{G}^{\mathsf{c}} \arrow[d, "\upharpoonright_{\mathsf{p}}"] \\
!^{\mathsf{c}}[\mathsf{q}];(\mathbb{S}).\mathsf{L}^{\mathsf{c}} \arrow[rr, "!\mathsf{pq}\mathbb{S}"] & & \mathsf{L}^{\mathsf{c}}
\end{tikzcd}$$

q receives:

$$\begin{tikzcd}
\mathsf{p} \to \mathsf{q} : (\mathbb{S}).\mathsf{G}^{\mathsf{c}} \arrow[r, "!\mathsf{pq}\mathbb{S}"] \arrow[d, "\upharpoonright_{\mathsf{q}}"] & \mathsf{p} \rightsquigarrow \mathsf{q} : (\mathbb{S}).\mathsf{G}^{\mathsf{c}} \arrow[r, "?\mathsf{qp}\mathbb{S}"] \arrow[dl, "\upharpoonright_{\mathsf{q}}"] & \mathsf{G}^{\mathsf{c}} \arrow[d, "\upharpoonright_{\mathsf{q}}"] \\
?^{\mathsf{c}}[\mathsf{p}];(\mathbb{S}).\mathsf{L}^{\mathsf{c}\prime} \arrow[rr, "?\mathsf{qp}\mathbb{S}"] & & \mathsf{L}^{\mathsf{c}\prime}
\end{tikzcd}$$

Tools for our LTS

Actions. $!\mathsf{pq}\mathbb{S}$ and $?\mathsf{qp}\mathbb{S}$

(Local) Environments. E such that, $E(\mathsf{p}) = \mathsf{L^c}_{\mathsf{p}}$ where $\mathsf{G^c} \upharpoonright^{\mathsf{c}} \mathsf{p}\ \mathsf{L^c}_{\mathsf{p}}$

Queues and Queue Environments. Q, buffers for asynchronous communication.

$$!^{\mathsf{c}}[\mathsf{q}];(\mathbb{S}).\mathsf{L^c} \xrightarrow{\quad\mathsf{step}\quad} \mathsf{L^c}$$

$$Q(\mathsf{p},\mathsf{q}) = [] \xrightarrow{\quad\mathsf{enqueue}\quad} Q(\mathsf{p},\mathsf{q}) = [\mathbb{S}] \xrightarrow{\quad\mathsf{dequeue}\quad} Q(\mathsf{p},\mathsf{q}) = []$$

$$?^{\mathsf{c}}[\mathsf{p}];(S).\mathsf{L^{c\prime}} \xrightarrow{\quad\mathsf{step}\quad} \mathsf{L^{c\prime}}$$

Tools for our LTS

Actions. $!\mathsf{pq}\mathbb{S}$ and $?\mathsf{qp}\mathbb{S}$

(Local) Environments. E such that, $E(\mathsf{p}) = \mathsf{L^c}_\mathsf{p}$ where $\mathsf{G^c} \upharpoonright^{\mathsf{c}}\mathsf{p}\ \mathsf{L^c}_\mathsf{p}$

Queues and Queue Environments. Q, buffers for asynchronous communication.

$$!^{\mathsf{c}}[\mathsf{q}];(\mathbb{S}).\mathsf{L^c} \xrightarrow{\quad\mathsf{step}\quad} \mathsf{L^c}$$

$$Q(\mathsf{p},\mathsf{q}) = [] \xrightarrow{\quad\mathsf{enqueue}\quad} Q(\mathsf{p},\mathsf{q}) = [\mathbb{S}] \xrightarrow{\quad\mathsf{dequeue}\quad} Q(\mathsf{p},\mathsf{q}) = []$$

$$?^{\mathsf{c}}[\mathsf{p}];(S).\mathsf{L^{c\prime}} \xrightarrow{\quad\mathsf{step}\quad} \mathsf{L^{c\prime}}$$

$$\begin{tikzcd}
 & \mathsf{G^c} \arrow[dl, "\upharpoonright^{\mathsf{c}}"'] \arrow[dr, "\upharpoonright^{\mathsf{q}}"] & \\
E & & Q
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\mathsf{G^c} \arrow[d, "\upharpoonright\upharpoonright"] \\
(E, Q)
\end{tikzcd}$$

Theorems

Theorem (Step Soundness)
If $\mathsf{G^c} \xrightarrow{a} \mathsf{G^{c\prime}}$ *and* $\mathsf{G^c} \restriction\, \restriction (E, Q)$*, there exist* E' *and* Q' *such that* $\mathsf{G^{c\prime}} \restriction\, \restriction (E', Q')$ *and* $(E, Q) \xrightarrow{a} (E', Q')$*.*

Theorem (Step Completeness)
If $(E, Q) \xrightarrow{a} (E', Q')$ *and* $\mathsf{G^c} \restriction\, \restriction (E, Q)$*, there exist* $\mathsf{G^{c\prime}}$ *such that* $\mathsf{G^{c\prime}} \restriction\, \restriction (E', Q')$ *and* $\mathsf{G^c} \xrightarrow{a} \mathsf{G^{c\prime}}$*.*

Theorem (Trace equivalence)
If $\mathsf{G^c} \restriction\, \restriction (E, Q)$*, then* $tr^g t \mathsf{G^c}$ *if and only if* $tr^l t (E, Q)$ *.*

Lemma, to give the flavour

$$
\begin{tikzcd}
p \to q :(\mathbb{S}).G^c \arrow[r, "!pq\mathbb{S}"] \arrow[d, "\upharpoonright_p"] & p \rightsquigarrow q :(\mathbb{S}).G^c \arrow[d, "\upharpoonright_p"] \\
!^c[q];(\mathbb{S}).L^c \arrow[r, "!pq\mathbb{S}"] & L^c
\end{tikzcd}
$$

$$
\begin{tikzcd}
p \to q :(\mathbb{S}).G^c \arrow[r, "!pq\mathbb{S}"] \arrow[d, "\upharpoonright_q"] & p \rightsquigarrow q :(\mathbb{S}).G^c \arrow[dl, "\upharpoonright_q"] \\
?^c[p];(\mathbb{S}).L^{c'} &
\end{tikzcd}
$$

Lemma, to give the flavour

$$
\begin{tikzcd}
p \to q :(\mathbb{S}).G^c \arrow[r, "!pq\mathbb{S}"] \arrow[d, "\upharpoonright_p"] & p \rightsquigarrow q :(\mathbb{S}).G^c \arrow[d, "\upharpoonright_p"] \\
!^c[q];(\mathbb{S}).L^c \arrow[r, "!pq\mathbb{S}"] & L^c
\end{tikzcd}
$$

$\longrightarrow$ Coq!

$$
\begin{tikzcd}
p \to q :(\mathbb{S}).G^c \arrow[r, "!pq\mathbb{S}"] \arrow[d, "\upharpoonright_q"] & p \rightsquigarrow q :(\mathbb{S}).G^c \arrow[dl, "\upharpoonright_q"] \\
?^c[p];(\mathbb{S}).L^{c'} &
\end{tikzcd}
$$

You Suffer...

- Formal proofs are not easy.
- Proof design is the key.
- Proof techniques are to be taken seriously: (co)induction, functions VS relations, treatment of bindings...

→ D. Castro-Perez, F. Ferreira, L. Gheri, and N. Yoshida. "Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes". PLDI 2021.
DOI: `https://doi.org/10.1145/3453483.3454041`
website: `http://mrg.doc.ic.ac.uk/publications/zooid-paper/`

→ This tutorial is available at `https://github.com/emtst/GentleAdventure`

… but Why?

Formal proofs are not easy

[1] Aydemir et al. "Mechanized Metatheory for the Masses: The POPLmark challenge." 2005

… but Why?

Formal proofs are not easy, but useful and fun!

As witnessed, e.g., by the influential POPLmark Challenge[1]…

[1] Aydemir et al. "Mechanized Metatheory for the Masses: The POPLmark challenge." 2005

... but Why?

Formal proofs are not easy, but useful and fun!

As witnessed, e.g., by the influential POPLmark Challenge[1]...

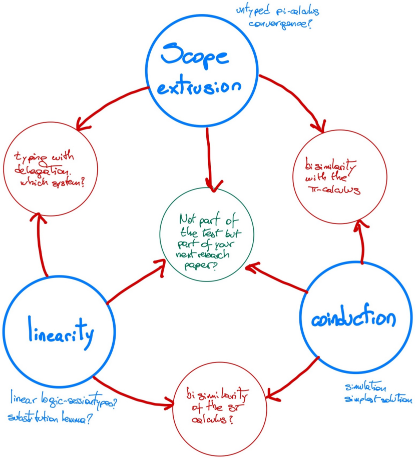
Towards a **Concurrent Calculi Formalisation Benchmark**

Challenge problems:

- name passing and scope extrusion
- linearity and behavioural type systems
- coinduction and reasoning about process algebras

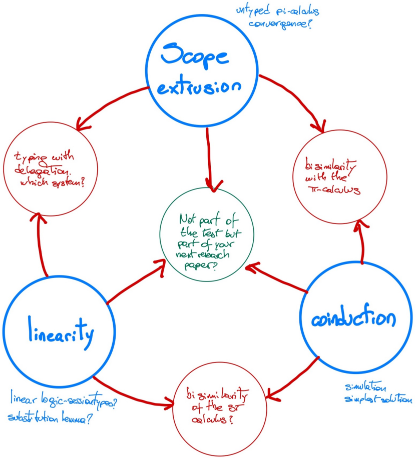
[1] Aydemir et al. "Mechanized Metatheory for the Masses: The POPLmark challenge." 2005

The Future is Unwritten... But Sketched!



- Concurrent Benchmark website:
  `https://concurrentbenchmark.github.io/`
- This tutorial:
  `https://github.com/emtst/GentleAdventure`

The Future is Unwritten... But Sketched!



- Concurrent Benchmark website: `https://concurrentbenchmark.github.io/`
- This tutorial: `https://github.com/emtst/GentleAdventure`

THANK YOU!